# Towards the static detection of erroneous executions in Ada 95[1]

F. Mazzanti, T. Marzullo*,

IEI-CNR, Via S.Maria 46, 56126 Pisa, Italy

June1996

**Abstract**

Being absolutely certain that a program execution will not result in unpredictable behaviour is very difficult. Although there is no way to avoid the underlying theoretical difficulties, we believe that a human-centered approach in exploiting advanced static analysis techniques is a viable solution. This paper presents some preliminary results of a feasibility study on this subject, which is currently underway at the I.E.I.

**Categories and Subject Descriptiors:** D.2.2 [Software Engineering]: Tools and Techniques; D.2.3 [Software Engineering]: Coding; D.2.4 [Software Engineering]: Program Verification; F3.1 [Logics and Meanings of Programs]:Specifying and Verifying and Reasoning about Progframs - *Mechanical verification*

## 1 Introduction

There is a class of errors (the so-called erroneous executions) which are particularly undesirable. In fact, when an erroneous execution occurs the program's behaviour becomes "unpredictable" and this means that:

a) No internal program error recovery will ever be able to restore the correctness of the execution;

b) The program execution might not simply be aborted as a consequence of the error, but might continue to produce unpredictably wrong interactions with the external environment.

Fortunately, reasonably written programs are unlikely to fail in this way. Erroneous executions in Ada95 are not easy to generate; usually a tricky sequence of operations is needed to observe an erroneous execution, and the required programming style is quite uncommon in real applications. From this point of view Ada is in a better position than many other languages, because many situations which are erroneous in other contexts are not so in Ada. For example, the use of the wrong array index in Ada causes an exception, while in C it may easily cause some unrecoverable memory overwriting. There are other languages which have adopted an even safer approach (see e.g. Java [1]), but this entails restricting language features and the way in which they can be controlled by the user.

---

[1] This paper is an updated version of a corresponding paper presented at the 9th International Software Quality Week, S.Francisco, U.S.A., May 21-24, 1996.

* C.N.R. Grant

One disadvantage of this "partially safe" approach is that a programmer might be misled into believing that erroneous executions never occur, whereas in fact they do occasionally occur and sometimes with a much higher probability than expected. Programmers should be helped in getting a complete and clear picture of the problem, but the Ada Reference Manual [2] requires a considerable amount of study to understand all the aspects involved.

Although erroneous executions are not common, being absolutely sure that they will definitely not occur is very difficult, but sometimes necessary.

High integrity systems are one such example, though a somewhat a border case: they are usually small in size, high costs are often acceptable for their production, and heavy restrictions in language usability may also be acceptable. However, this is also true of software which aims to be of good quality. We are all ready to accept (non-critical) software which is not perfect, but when a program behaves in an erroneous way the sign of bad quality becomes evident.

When we need to be absolutely sure that no erroneous execution will occur, the current solution is:

- To make use of a very drastic subset allowing, for example, a complete formal verification of the system (see[3] [4] [5] for some Safe Ada subsets).
- To rely on code inspections (made by humans).
- To rely on informal design and coding guidelines tailored to the reduction of this danger.

Unfortunately, none of these solutions is fully satisfying because:

- Drastic subsets also reduce the language's usability (sometimes in a unacceptable manner).
- Human code inspections are expensive and prone to failures: they cannot give any certainty.
- When design and coding guidelines tend to state generic programming principles they may not give sufficiently detailed advice about what and what not to do. Moreover, there is the danger that they simply reflect the current trend in "expert opinions" without being backed up by strong scientific arguments [6] [7]. But, most importantly, if they are not mechanically enforced/verified, deviations from them can only be detected by human inspections with all the consequent limits and costs.

The ideal solution would be to use a specific quality assurance tool to achieve the desired level of confidence on the predictability of the program. Unfortunately no tool of this kind is currently available. Quality checkers often deal with formatting issues, sometimes data flow analysis, and occasionally some form of task synchronization analysis (e.g. deadlock detection): some checks are performed, but without the necessary completeness to be able to state that no erroneous executions will ever occur even when tasking is widely used and optimizations not turned off (see for example [8] for a list of what is available).

Two events have occurred in the last two years that have prompted us to set up a small project aimed at investigating the feasibility of the static detection of these erroneous executions.

The first event was the completion of the 1995 revision of the Ada standard (ANSI/ISO/IEC-8652:1995). In the 1983 version of the language (ANSI/MIL-STD 1815A) erroneous executions were very difficult to understand because of the many deep ambiguities in the reference manual (see for example [9], [10] for a survey). With the new Ada95 standard, the situation from the point of view of the language definition is much improved: almost all the uncertainties of the previous version have been eliminated, though some loosely described aspects still remain.

The second event was the production of the first public domain Ada compiler (GNAT [11]), developed at NYU, whose sources are distributed according to the GNU general public licence.

Any attempt to build an advanced static analysis tool requires access to the type of information typically collected by the compiler front-end. Because of the language's complexity, rebuilding from scratch such a front-end is a not an affordable task for most people. Moreover, ASIS interfaces [12] (a standard Ada library for compiler interfacing) are not frequently supported by commercial compilers and still require expensive Ada environments to be bought. Now, however, we have the opportunity to directly reuse the sources of the gnat front-end, extending them with all the checks that are appropriate for our analysis[2].

# 2 A human centered approach

The mechanical detection of all and only those programs which will lead to an erroneous execution is, even theoretically, an impossible task. Figure 1 shows a class of programs whose execution is erroneous if and only if a called subprogram terminates. If we were able to mechanically detect the erroneousness of all programs in this class, we would also be able to mechanically prove the termination of any program.

```
with Other_Main;
procedure Main is
begin
   Other_Main;
   <perform something erroneous>
end Main;
```

Figure 1- a scheme of potentially erroneous programs

In implementing our checking strategies, we have only two possibilities. We can try to recognize just a subset of all the erroneous programs, or try to label as "suspect" a superset of all the erroneous programs (i.e. a set that also includes some programs that are actually correct). We are interested in the second alternative, which basically consists in finding a set of mechanically verifiable rules that guarantee the detection of (at least) all the erroneous programs.

A simple solution would be to remove from the Ada grammar all the constructs that might lead to an erroneous execution. The result is the immediate removal of access types,. discriminated types, exceptions, and tasks (just to mention some). In fact, this is the direction towards the above mentioned "strict safe Ada subset", which we believe to be unsatisfactory.

A more complex solution would be to preserve all the existing Ada features, and try to verify that they are used safely. This can be done by analysing the programmers' behaviour, looking at the reasoning that convinces them of the correctness of their code, and exploiting this knowledge and deduction power in our analyser. This implies a more "human centered" approach, which tries to identify all the ways that certain critical language aspects are safely used in practice, while disallowing the other types of uncommon and unsafe uses.

The above approach is by nature incremental, since the knowledge of the "safe patterns of use" is likely to grow in time and with the experience gained.

# 3 Detecting erroneous executions

The Ada language definition allows the occurrence of erroneous executions in several ways:

---

[2] Currently, a free ASIS interface is being developed for GNAT (see the project description at the URL "http://www.acm.org/sigada/WG/asiswg/asiswg.html"). When this product will become available it will probably constitute the best approach for the development of new quality assurance tools.

1) The occurrence of certain types of errors make the program execution directly erroneous.
2) Certain well-defined situations may cause a variable to get an "abnormal value"; any subsequent use of this variable (except when a new value is assigned to it) is then erroneous.
3) Under certain circumstances the execution of a construct has an "unspecified effect" (or an "unspecified behaviour", or an "unspecified semantics"). In these cases the effect is implementation dependent and possibly erroneous (the implementation is not even required to document its particular choices).
4) Sometimes the execution of a construct has an "implementation-defined effect" (or behaviour, or semantics). This case simalar to the previous one, with the only difference that the implementation is now required to document the particular choices which have been made.

Actually, erroneous executions might also occur when some undefined aspect of the language is hit. With "undefined aspect" in this case we really mean "holes" in the language definition, not aspects which have been explicitly labelled as "unspecified" or "implementation defined". These are (hopefully very rare) true bugs in the language definition and appropriate action is likely to be take by the appropriate maintenance committee whenever any of them is found[3].

Getting a detailed and complete picture of how and when an erroneous execution is generated requires a certain amount of effort in digesting the Reference Manual [2] and other related documents such as the Rationale [13], the Annotated Manual [14] and some Language Studies [15]). It is beyond the scope of this paper to detail the resulting full picture. Appendix A outlines a few tables which summarise the most important preliminary results of a detailed study carried out at IEI [16], and to which we refer.

As we can see from Table 2 in Appendix A, erroneous executions are often related to very specialized aspects of programming, such as the development of hardware/software interfaces, or the need to handle directly, using dangerous programming techniques, the program resources such as memory allocation, and task descriptors.

In these cases the danger of using these critical features is very explicit, and often simply giving the programmer the list of all the points where this happens is enough (without even trying to identify safe patterns of use for these features).

However, there are some cases of erroneous executions which are not related to any kind of specialized programming need. They are simply some "dangerous spots" related to the type system, or to tasking issues, or to the input-output system, or to the possibility of optimizations (which may even be implicit).

In these cases it is really important that the Reference Manual describes these aspects clearly and completely, and that mechanically verifiable "safe patterns" of use are found, so that appropriate verification can be carried out.

Table 4 in Appendix A classifies the various cases of erroneous execution with respect to the clarity in which they are presented in the language manual. On the other hand, Table 5 draws some preliminary conclusions on the difficulty of finding a "safe pattern of use". Of the seven cases of erroneous execution related to generic programming aspects (i.e. not related to specialized programming needs), only two are described unsatisfactorily (cases 1 and 5), and only one (case 5) still does not have a fully satisfying solution in terms of "safe patterns of use".

The following subsections illustrate several sample cases of erroneous executions taken from the main study mentioned before. For these examples we will show the difficulty / usefulness of identifying safe patterns of use, and the type of verification that we would need to perform.

---

[3] Information on how to submit comments on the standard is given in the "Introduction" section of the Reference Manual .

## 3.1 Use of a subcomponent which depends on a discriminant

In this section we present a dangerous spot in the Ada type system. The danger is related to the existence of unconstrained record variables, i.e. variables whose internal structure depends on the run-time value of the record discriminant. In Ada the correctness of the accesses to the variable components is automatically checked at run-time (unlike other languages, like C, in which this kind of check is left to the programmer) every time the name of a record component is evaluated.

Unfortunately this kind of check does not totally guarantee the correctness of all the possible uses to the record component. The possibility of an erroneous execution is stated explicitly (in a precise but rather enigmatic form) in Section 3.7.2(4) of the Reference Manual.

[RM 3.7.2(4) *The execution of a construct is erroneous if the construct has a constituent that is a name denoting a subcomponent that depends on discriminants, and the value of any of these discriminants is changed by this execution between evaluating the name and the last use (within this execution) of the subcomponent denoted by the name.*

The problem appears when the name of a component of the record variable is successfully evaluated but, before the denoted subcomponent is actually used, the whole structure of the unconstrained variable is changed by a new complete assignment.

Figure 2 shows an example of how this might occur:

```
type Int_Ptr is access Integer;

type Union(Tag:Boolean:= True) is record
  case Tag is
    when True  => Value: Integer;
    when False => Ptr: Int_Ptr;
  end case;
end record;

Var: Union := (True,10);    -- (1)

function Change_Var return Integer is
begin
  Var:= (False,null);       -- (3)
  return 777;
end Change_Var;

Var.Value:= Change_Var;     -- (2,4)
```

Figure 2- erroneous use of record subcomponent

(1) We have an unconstrained record variable ("Var") , initially with an Integer component ("Value").

(2) When an assignment to the record component "Var.Value" is executed, one possibility is that the target of the assignment is evaluated before the expression on the right hand side. In this case the name "Var.Value" is evaluated successfully since the Integer component "Value" is precisely the one existing at this time. The evaluation of the name of the record component returns the address of the subcomponent which will later be used for the update.

(3) During the evaluation of the expression on the right hand side of the assignment we call a function ("Change_Var") whose side effect is to completely reset the value of the

unconstrained variable ("Var") by substituting the integer component ("Value") with a pointer component ("Ptr").

(4) When the value returned by the function is used for the update, the original "Value" component no longer exists, and it is very likely that the integer value is written over the "Ptr" component of the record variable. This makes the execution erroneous.

No erroneous execution would have occurred if the expression on the right hand side of the assignment had been evaluated before the variable name on the left hand side (but following this ordering is optional and not always possible).

According to the Reference Manual an erroneous execution occurs when:

- A construct makes use of a "name denoting a subcomponent that depends on discriminants"
- The subcomponent is used some time after the name has been evaluated
- In the time between the evaluation of the name and the use of the subcomponent the discriminant of the variable is changed.

A full understanding of the consequences of this clause is rather difficult because nowhere does the Reference Manual summarise the possible cases of "constructs" which make this kind of "deferred use" of a name. Other documentation sources have to be analysed in order to understand the real impact of this case of erroneous execution, such as the commentaries on the Ada83 standard [17] or the Annotated version of the Ada 95 Reference Manual [14].

Besides problems of interpretation (discussed in detail in [16]), we are also interested in seeing whether it is possible to identify a "safe pattern" of use of record components which depend on a discriminant, which does not unnecessarily restrict the usability of the construct, and which is also mechanically verifiable.

In particular, we have no intention of forbidding the use of discriminants, or unconstrained types, or to make the absence of side-effects and aliasing for subprograms mandatory. On the contrary, we want to check as closely as possible that the situation described in Section 3.7.2.(4) of the Reference manual does not occur.

Let us now consider again the final assignment statement of Figure 1. One of the possible strategies to prevent the risk of an erroneous execution might consist in reporting as being suspect any assignment whose left hand side denotes a subcomponent of an unconstrained record, and whose expression in the right hand side contains a function call, regardless of whether it performs a side-effect or not. This is like saying that the "safe pattern" of use consists in using only variable names or literals as expressions that can be assigned to a subcomponent of an unconstrained record which depends on some discriminant.

This simple strategy is sufficient to prevent erroneous executions, with very few overheads imposed on the programmer. In order to pass this kind of checking the fragment of code shown in Figure 2 should have been rewritten as shown in Figure 2b: all the side effects previously performed during assignment (3), are now performed before the beginning of assignment (2), and when the "Var.Value" name is evaluated an exception is raised because that component no longer exists.

The overhead imposed on the programmer is acceptable, but probably only if the "coding guideline" corresponding to the "safe pattern" was already known to the programmer before writing the code.

If we plan to verify the predictability of a program developed without prior knowledge of this coding guideline a lot of programs which are actually correct are very likely to be reported as being "potentially erroneous". For example, any assignment of the form shown in Figure 3 should be considered unsafe (because we are calling function "+" on the right hand side of the assignment).

A better solution, in this case, is to verify that the evaluation of the expression in the right hand side of the assignment does not produce any side-effects on the target of the assignment.

This type of check might entail a program-wide analysis of the syntactic structure of all the functions used in the right-side expression, and of all the subprograms and tasks directly or indirectly called by these functions. Again, this kind of checking can be done with increasing levels of complexity, but for this particular case of erroneous execution a reasonable compromise can certainly be found.

```
type Int_Ptr is access Integer;

type Union(Tag:Boolean:= True) is record
  case Tag is
    when True  => Value: Integer;
    when False => Ptr: Int_Ptr;
  end case;
end record;

Var: Union := (True,10);    -- (1)
Tmp:Integer;

function Change_Var return Integer is
begin
  Var:= (False,null);       -- (3)
..return 777;
end Change_Var;

Tmp := Change_Var (2);
Var.Value:= Tmp;     -- (4)
```

Figure 2b: A safe use of a record subcomponent

```
...
X:Integer;
...
Var.Value:= X+1;
```

Figure 3 A potentially unsafe assignment

## 3.2 Exceptions and Optimizations

Another (often unfamiliar) critical aspect of Ada is related to the freedom given to implementations in performing optimizations. A first class of optimizations are those aimed at improving the way the cpu or memory resources are used, but such that the set of external interactions produced by the execution of a program is still consistent with the requirements of the language definition if no optimizations have been carried out (the so-called "canonical semantics"). These are somewhat "transparent" optimizations and do not create any problems.

However, a second class of optimizations is allowed, which may have a greater effect on program behaviour and which may easily cause a program execution to become erroneous. This is the class of optimizations described in Section 11.6 of the Reference Manual. We are not going to describe this aspect exhaustively, because it is not easy to understand the underlying motivations and consequences (see [19], [20] for more detailed discussions).

Below we will show a few examples which highlight the main points, the difficulties, and the possible solutions.

One of the optimizations that Ada95 allows is, for example, the parallelization of the cycle shown in Figure 4. In this case, the canonical semantics of the language does not allow us to

take advantage of a parallel architecture because, should an exception be raised (caused by a division by zero, for example) when computing the new value for element K, only the elements from 1 to K-1 are required to be updated, leaving the others untouched. In other words, before making an assignment to item K, we should be sure that the assignment to item K-1 did not raise an exception. However, this requirement defeats any attempt to perform all the assignments in parallel.

```
subtype Index is Integer range 1..Max;
Target, Base: array (Index) of Integer
...
declare
begin
   ...
   for I in Index'Range loop
    Target(I):= Target(I)/Base(I);
   end loop;
   ...
exception
   when Costraint_Error => null;
end;
```

Figure 4 A potentially parallel cycle

Section 11.6 of the Reference Manual relaxes the behaviour required by the canonical semantics requiring that, should a predefined check fail during the execution of the cycle, then:

- The external interactions of the program need only reflect the fact that an exception has been raised inside the block, but not necessarily at the same point required by the canonical semantics (in our case the exception might appear to be raised at the end of the cycle instead of in the middle of it).
- All the variables external to the block which have been updated during the execution of the block before raising the exception are allowed to become abnormal.

These rules allow us to perform all the assignments in parallel, possibly raising an exception at the end of the cycle, in which case the "Target" vector would be filled with abnormal values. If no exceptions are raised, the computation is performed efficiently on parallel architectures, whereas if an exception is raised, the entire results should be discarded.

Another important type of optimization is aimed at exploiting pipelined hardware architectures (like some RISC machines). In order to achieve maximum performance on these machines is it sometimes necessary to bring forward the beginning of the execution of an instruction, when the previous one has not yet been completed, or to slightly reorder the sequence of instructions otherwise generated by the compiler.

```
X,Y: Integer;
...
declare
begin
   ...
   Y:= 1111;
   X:= 1000/X;
   ...
exception
   when Costraint_Error => null;
end;
```

Figure 5 A potentially reordable sequence of operations

For example, in the case shown in Figure 5, part of the evaluation of the expression "1000/X" may be brought forward with respect to the completion of the execution of the assignment "Y:=1111", so that, even if an exception is raised by the second statement (division by zero), the other variable Y might result only partially (or not at all) updated. The rules in Section 11.6 of the Reference Manual allow also this type of optimization. However, should a predefined check fail during the execution of the sequence of statements of the block, all the (non local) updated variables (i.e. Y) might become abnormal.

The consequences of Section 11.6, from the point of view of the erroneous execution are clearly related to the possibility of generating abnormal values. Figures 6 and 7 show two examples of potentially erroneous programs corresponding to the situations shown in Figures 4 and 5:

```
subtype Index is Integer range 1..Max;
Target, Base: array (Index) of Integer
...
declare
begin
  ...
  for I in Index'Range loop
   Target(I):= Target(I)/Base(I);
  end loop;
  ...
exception
  when Constraint_Error =>
    for J in Index'Range loop
     Put(Target(I));
    end loop;
end;
```

Figure 6 A potentially erroneous program

```
X,Y: Integer;
...
declare
begin
  ...
  Y:= 1111;
  X:= 1000/X;
  ...
exception
  when Costraint_Error => Put(Y);
end;
```

Figure 7 A potentially erroneous program

Note that the above two examples were constructed with great effort starting from the wording of Section 11.6 in the Reference Manual, because of the extremely obscure presentation of the problem which is made in the standard. Some official clarifications are needed for this section of the manual, which already in the 1983 version was well-known for its ambiguity and vagueness. It is also curious that the whole chapter related to Section 11.6 of the reference manual is completely omitted from the Ada Rationale [13], leaving the (probably wrong) impression that also the language designers had difficulty in finding an exhaustive and clear picture of the problem and its solution.

From the point of view of the identification of a "safe pattern" of use of optimizations, the idea of completely disallowing them (which might still be an acceptable solution for certain

critical systems) is not satisfactory for us. The optimal solution of the problem is to statically verify that no language-defined check ever fails during program execution.

By investing in complex data-flow analysis, and exploiting all the Ada "safe" features (e.g. subtyping) we might hope to be able to verify that most fragments of code are actually safe from this point of view (see [21] and [22]). However, we believe that the problem in its generality is too complex to be completely solved in such a desirable way.

Another solution might be to verify that all the non-local variables updated by a sequence of statements covered by an exception handler, are reset to clean values when a predefined exception is raised. This kind of checking is easier to perform mechanically, although it does require a program-wide analysis of all the subprograms and tasks directly or indirectly activated by the execution of the sequence of statements.

The above type of checking might be further relaxed for those non-local variables that we can prove are no longer used by the program (for example, because the exception if raised again to a level at which the variable is no longer accessible), or are reset before use in other parts of the program, or are checked for validity before being used.

## 3.3. Unsynchronized accesses to unprotected variables

Misuse of tasking is another potential source of erroneous executions in Ada. In this case the danger is related to the use of unprotected non-local variables (i.e. variables directly used by more than one task, but not protected against concurrent accesses).

In particular, in Section 9.10(11) the Reference Manual states:

[RM 9.10(11)] *Given an action of assigning to an object, and an action of reading or updating a part of the same object (or of a neighboring object if the two are not independently addressable), then the execution of the actions is erroneous unless the actions are sequential.*

We must say that in Ada83 this aspect was presented in an extremely ambiguous way and that the new language revision has done a wonderful job in stating precisely when this kind of error might occur, and in reducing the need for such a dangerous feature with the introduction of protected types (i.e. data types protected against concurrent accesses).

As usual, we do not wish to jump to the simple conclusion that unprotected non-local variables should not be used. On the contrary, although the need for them has now been greatly reduced, we are interested in observing how difficult it is to perform a mechanical verification of the fact that they are used correctly.

Clearly, there are some safe patterns of use which should be allowed (e.g. when a non-local variable is used only by one task). Checking for slightly more complex types of safe patterns can be a little harder, but still feasible (e.g. a shared variable is used by one task before starting the activation of a second task, by the execution of this second task, and by the first task again after the termination of the second task). Unfortunately, as the patterns of use become less constraining, for example involving entry-calls or task objects created by allocators, the complexity of the checks which would be needed becomes very high and requires a sophisticated analysis of the concurrent behaviour of the program.

The verification of the properties of concurrent (Ada) programs is a very active research field. Many approaches have been investigated (typically for deadlock analysis) by using Petri-Nets, process algebras, temporal logic, and many others (see e.g. [23], [24], [25],[26], [27], [28]).

We are not aware of any specific attempt to prove the correctness of the code from the point of view of the uses of unprotected shared variables, but we have no reason to believe that the same analysis techniques could not also be used for the type of verification we are interested in.

In conclusion, this is one of the points for which we still do not have a satisfying mechanically verifiable solution. Further investigations are thus needed and promising results can probably be expected.

### 3.4 Invalid values returned by imported procedures

The example discussed in this section is completely different from the previous ones. First of all, it is not related to some generic programming aspect like the use of unconstrained types, the possibility (for the compiler) of performing optimizations, or the use of tasking. On the contrary, this example it is related to Section B of the Reference Manual: a section describing features for writing mixed-language programs. Secondly, in this case we present an example with a "negative" result, i.e. an example of a construct for which no "safe pattern" of use can probably be found.

What we want to highlight with this example is that even a negative result can sometimes be a fully satisfying solution.

In this case we deal with variables (of a non scalar type) passed as "in out" or "out" parameters to an imported procedure (i.e. a procedure written in another language linked together with the Ada program).

In Figure 9 we show a possible Ada interface towards such an imported program.

```
package My_Interface is
  type Int_Ptr is access Integer;

  procedure Initialize (My_Ptr: out Int_Ptr);

private
  pragma Import(C,Initialize);
end My_Interfaces;
```

Figure 9 A potentially erroneous program

The problem occurs if, after returning from the procedure, the representation of the parameter does not represent a value of the type Int_Ptr, in which case the object used as the actual parameter becomes abnormal. Nothing can probably be done from the point of view of the static verification of an Ada program, to ensure that no abnormal values of this kind are produced by the Fortran routine. In this case, all a verification / quality assurance tool can do is to report the presence of this kind of potentially unsafe code, allowing a human reviewer to verify the interface and take responsibility for the correctness of the program.

However, the simple mechanical identification of all the potentially unsafe points of this kind is still a useful, though partial, result.

## 4 Conclusions

The IEI project JADA [27] is an internal project that began in 1995. So far we have conducted a review of Ada (from the point of view of erroneous executions) and a preliminary analysis of the possible "safe patterns of use". We have also started a prototypal implementation of a program analyser by extending the front-end of the GNAT [11] Ada compiler . All these activities are still in their early stages and need much more work.

One future activity will be to validate the approach by applying the tool to several "real" projects. We will thus be able to verify how much the identified "safe patterns of use" are really "human centered", and start a new cycle of our incremental approach to predictability analysis.

The overall framework is a wider study on the set of minimal properties an Ada program should satisfy it order to be reasonably analysable with formal techniques. If no assumptions are made, in fact, the formal program semantics is likely to be so complex that building and working with it easily becomes a task of untreatable complexity (see e.g. [28]). We want to be able to associate a reasonably simple semantics to any reasonably well written program, to be able to understand by ourselves this formal semantics, and to be able to effectively use it to prove partial correctness and safety properties of the code (in particular, within our verification environment [18]).

There are four aspects which we believe it would be extremely important to be able to check mechanically.

- Absence of erroneous executions
- Absence (or safely controlled use) of language defined exceptions
- Absence of unnecessary nondeterminism and
- Independence from implementation-dependent aspects.

Only the successful treatment of these four points would allow us to develop a simple and usable semantics for Ada. However, even without considering formal analysis issues, these properties are of very wide interest, and the static mechanical verification of them would be a major positive result for all Ada developers.

The expected benefits of our research are much greater than the set of checks we will actually be able to perform. In particular, an indirect result of our effort, is the definition of a programming style aimed at improving program predictability and robustness : a programming style that maximises the possibility of static checks while still maintaining language usability. A detailed analysis of the difficulties encountered is expected to provide a sound basis for the definition and evaluation of "safe subsets", "style guides", "programming paradigms", "checklist". What can be learned from this project (also in terms of what could not be achieved) is probably as valuable as the production of the tools itself.

# 5 Acknowledgements

We are grateful to our colleagues for the discussions on the subject, and in particular to Stefania Gnesi, Alessandro Fantechi and Carlo Manconi. .

# 6 References


[1] Sun Microsystems "The Java Language: A White Paper" <http://java.sun.com/doc/overview/index.html>

[2] ISO -International Standard ANSI/ISO/IEC-8652:1995, "Ada 95 Reference Manual" <http://lglwww.epfl.ch/Ada/LRM/9X/rm9x/rm9x-toc.html>

[3] Carrè B.; Garnsworthy J,; Marsh W. "SPARK: A Safety-Related Ada Subset" Proceedings of *Ada in Transition*, Ada UK International Conference, 1992, London Docklands.

[4] B.Jepson -"A Study of High Integrity Ada - Language Review" -York Software Engineering and British Aerospace Defence Limited, 1993, Mod Contract Number: SLS31c/73, document SLS31c/73-1-D , <ftp://minster.york.ac.uk/YSE/hia-wp1.ps.Z>

[5] Smith M. K. "The AVA Reference Manual: Derived from ANSI/MIL-STD-1815A-1983" Computational Logic Inc. 1992.

[6] Fenton N "How effective are software Engineering methods?" *J. Systems and Software* n. 22, pp 141-146, 1993

[7] F.Mazzanti "Coding Regulations for Safety Critical Software Development" Proceedings *Second IEEE International Software Engineering Standards Symposium* (ISESS'95), August 1995, Toronto Canada.

[8] AIC list of commercial tools: <http://sw-eng.falls-church.va.us/AdaIC/tools/tool.txt>

[9] Wichmann B.A. "Insecurities in the Ada programming language: An interim report" NPL Report DITC 137/89, January 1989.

[10] F.Mazzanti "Ada erroneous executions: the current status of definition" Deliverable I of task 4.6 of CEC Multi-Annual Project n. 755: "SFD-APSE" 1988.

[11] E.SChonberg. B. Banner: "The GNAT Project: A GNU-Ada 9X Compiler" Proceedings of *Tri-Ada '94,* Balimore, Maryland, 1994 (see also <ftp://cs.nyu.edy/pub/gnat>).

[12] J.Bladebladen et al. "Ada Semantic Interface Specification (ASIS)", Proceedings of *Tri-Ada '91*, San Jose, California, 1991
(see also <http://www.acm.org/sigada/WG/asiswg/asiswg.html>)

[13] AdaIC "Ada 95 Rationale"
<http://lglwww.epfl.ch/Ada/LRM/9X/Rationale/rat95html/rat95-contents.html>

[14] AdaIC "Annotated Ada 95 Reference Manual"
<ftp:///public/AdaIC/standards/95lrm_rat/v6.0/aarm.doc>

[15] AdaIC "Language Study Notes for Ada95" <ftp://public/AdaIC/standards/95lsn>

[16] Mazzanti,Marzullo: "Guide to Unpredictabilities in Ada 95", *IEI Technical Report* B4-20, March 1996

[17] AdaIC "Ada 83 Commentaries" <ftp://public/AdaIC/standards/83com>

[18] A.Bouali, S.Gnesi, S.Larosa "The integration project for the JACK Environment" *Bullettin of the EATCS*, n.54, October 1994, pp 207-223
(see also <http://rep1.iei.pi.cnr.it/Projects/JACK/part1_birdeye/whatisjack.html>)

[19] Language Precision Team - D. Guaspari, J.McHugh, W.Polak, M.Saaltink "Toward a Formal Semantics for Ada9X" Final Report NAS1-18972 Task 11,October 31,1994 <ftp://ftp.oracorp.com:/pub/LPT/oct94.ps>

[20] AdaIC "on Optimization of Use of Scalar variables" *Language Study Note* LSN-1066

[21] B.Winner "Investigation of implicitly raised Ada predefined exceptions" Proceedings *Tenth Annual Washington Ada Symposium* (WADS'93) p.71-79, Mclean, VA, USA, July 1993.

[22] B.Scaefer, "Static analysis of exception handling in Ada", *Software Practice and Experience*, Vol 23, Iss. 10, pp 1157-74, October 1993.

[23] M.B. Dwyer, L.A.Clarke, K.A.Nies " A Compact Petri Net Representation for Concurrent Programs", Proceedings of 17th *International Conference on Software Engineering*, Seattle, Washington, USA, April 23-30, 1995

[24] W.Y.Joung "Re-designing tasking structures of Ada programs for analysis: A case study" *Software Testing, Verification and Reliability*, Vol. 4, Iss.4, pp 223-53, Dec 1994

[25] M.Young,R. Taylor "Combining static concurrency analysis with symbolic execution" *IEEE Transactions on Software Engineering*, 14(10):1499-1511, Octore 1988.

[26] W.E.Howden, G.M. Shi "Linear and structural event sequence analysis" *ISTA'96*, San Diego, Ca, USA 1996

[27] F.Mazzanti "Jada Overview" <http://rep1.iei.pi.cnr.it/projects/jada/>

[28] R.Taylor "Complexity of analyzing the synchronization structure of concurrent programs" *Acta Informatica*, 19:57-84, 1983.

# Appendix A: Summary of erroneous executions in Ada 95

**TABLE 1** **Index of unpredictabilites**

| *Short description* | *RM References* |
|---|---|
| **1)** Use of subcomponents which depend from a discriminant | 3.7.2 (4) |
| **2)** Disrupting of an assignment because of an abort | 9.8 (21) |
| **3)** Unsynchronized updates of unprotected variable | 9.10 (11-14) |
| **4)** Suppression of checks | 11.5 (26) |
| **5)** Exceptions and optimizations | 11.6 (4-6) |
| **6)** Alignment and address clauses | 13.3 (27) |
| **7)** Machine code insertions | 13.8 |
| **8)** Invalid values returned by imported precedures | 13.9.1 (8) |
| **9)** Use of pragma Restrictions | H4(26,27) |
| **10)** Unchecked programming | 13.9.1(12,13), 13.11.2 (16) |
| **11)** Storage management | 13.11 (21),H.2 (1) |
| **12)** Read operations | 13.13.2(35), A.13(17) |
| **13)** I/O for access types | A.7 (6) |
| **14)** Operations on deleted or closed default files | A.10.3 (22-23) |
| **15)** Interfaces to other languages | B |
| **16)** Interface to C.strings | B.3.1 (51-57) |
| **17)** Interface to C.pointers | B.3.2 (35-42) |
| **18)** Priority of inturrupts | C.3.1 (14) |
| **19)** Use of task ids | C.7.1 (18), C.7.2 (14-15), D.4(12), D.11 (9),D5(12) |
| **20)** Invalid addres representation clauses | 13.3(13, 27,28) |
| **21)** Storage management | 13.11 (20) |

*** implementation-defined aspects are not considered in this table ***

**TABLE 2** **Affected language aspect**

| *Generic programming aspects* | | *Specialized programming aspects* | |
|---|---|---|---|
| Types: | 1) | Unchecked programming: | 4) 9) 10) 11) |
| Concurrency: | 2) 3) | Real-Time Programming: | 19) |
| Optimizations: | 5) | Hardware interfaces: | 6) 7) 18) 20) 21) |
| Input-Output: | 12) 13) 14) | Software interfaces: | 8) 15) 16) 17) |

**TABLE 3** **Relation with erroneous execution**

| | |
|---|---|
| Causes erroneous execution: | 1) 3) 4) 6) 7) 9) 10) 11) 14) 15) 16) 17) 18) 19)20) |
| Causes Abnormal Value: | 2) 5) 8) 12) |
| Has unspecified effect: | 13) 21) |

**TABLE 4** **How clearly and exhaustively the RM describes the aspect**

| | |
|---|---|
| Rather clearly and exhaustively: | 2) 3) 4) 6) 8) 9) 11) 12) 13) 15) 16) 17) 18) 19) |
| Need some minor interpretation: | 10) 14) 20) 21) |
| Need some major interpretation: | 1) 5) |
| Not explicitly mentioned: | 7) |

**TABLE 5** **Has a "safe pattern of use" (mechanically verifiable) been found which does not compromise the usability of the construct?**

| | |
|---|---|
| Yes: | 1) |
| Yes, but the solution can be improved: | 2) 3) 5) 14) |
| Not, but probably it is not worth the effort: | 6) 7) 8) 12) 13) 15) 16) 17) 18) 19) 20) |
| Not yet, more studies are needed: | 4) 9) 10) 11) 21) |